Chapter 3

Hardware Shading Effects

Wolfgang Heidrich

Real-time Shading: Hardware Shading Effects

Wolfgang Heidrich
The University of British Columbia

Abstract

In this part of the course we will review some examples of shading algorithms that we might want to implement in a real-time or interactive system. This will help us to identify common approaches for real-time shading systems and to acquire information about feature sets required for this kind of system.

The shading algorithms we will look at fall into three categories: realistic materials for local and global illumination, shadow mapping, and finally bump mapping algorithms.

1 Realistic Materials

In this section we describe techniques for a variety of different reflection models to the computation of local illumination in hardware-based rendering. Rather than replacing the standard Phong model by another single, fixed model, we seek a method that allows us to utilize a wide variety of different models so that the most appropriate model can be chosen for each application.

1.1 Arbitrary BRDFs for Local Illumination

We will first consider the case of local illumination, i.e. light that arrives at objects directly from the light sources. The more complicated case of indirect illumination (i.e. light that bounces around in the environment before hitting the object) will be described in Section 1.3.

The fundamental approach for rendering arbitrary materials works as follows. A reflection model in reflection model in computer graphics is typically given in the form of a *bidirectional reflectance distribution function* (BRDF), which describes the amount of light reflected for each pair of incoming (i.e. light) and outgoing (i.e. viewing) direction. This function can either be represented analytically, in which case it is called a reflection model), or it can be represented in a tabular or sampled form as a four-dimensional array (two dimensions each for the incoming and outgoing direction).

The problem with both representations is that they cannot directly be used in hardware rendering: the interesting analytical models are mathematically too complex for hardware implementations, and the tabular form consumes too much memory (a four-dimensional table can easily consume dozens of MB). A different approach has been proposed by Heidrich and Seidel [9]. It turns out that most lighting models in computer graphics can be factored into independent components that only depend on one or two angles. These can then be independently sampled and stored as lower-dimensional tables that consume much less memory. Kautz and McCool [12] described a method for factorizing BRDFs given in tabular form into lower dimensional parts that can be rendered in a similar fashion.

As an example for the treatment of analytical models, consider the one by Torrance and Sparrow [29]:

$$f_r(\vec{l} \rightarrow \vec{v}) = \frac{F \cdot G \cdot D}{\pi \cdot \cos\alpha \cdot \cos\beta}, \tag{1}$$

where f_r is the BRDF, α is the angle between the surface normal $\vec{n}$ and the vector $\vec{l}$ pointing towards the light source, while β is the angle between $\vec{n}$ and the viewing direction $\vec{v}$. The geometry is depicted in Figure 1.

For a fixed index of refraction, the Fresnel term F in Equation 1 only depends on the angle θ between the light direction $\vec{l}$ and the micro facet normal $\vec{h}$, which is the halfway vector between $\vec{l}$ and $\vec{v}$. Thus, the Fresnel term can be seen as a univariate function $F(\cos\theta)$.

The micro facet distribution function D, which defines the percentage of facets oriented in direction $\vec{h}$, depends on the angle δ between $\vec{h}$ and the surface normal $\vec{n}$, as well as a roughness parameter. This is true for all widely used choices of distribution functions, including a Gaussian distribution of δ or of the surface

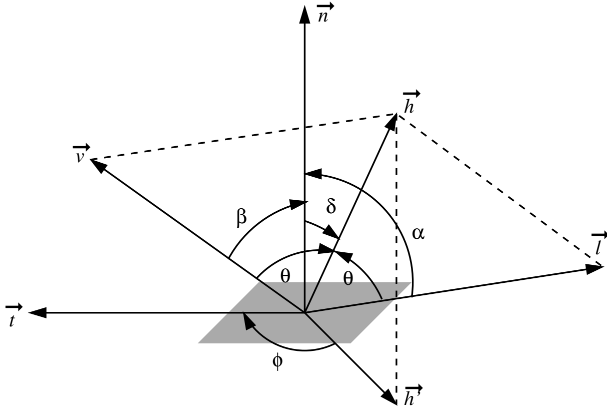

Figure 1: The local geometry of reflection at a rough surface.

height, as well as the distribution by Beckmann [3]. Since the roughness is generally assumed to be constant for a given surface, this is again a univariate function $D(\cos\delta)$.

Finally, when using the geometry term G proposed by Smith [27], which describes the shadowing and masking of light for surfaces with a Gaussian micro facet distribution, this term is a bivariate function $G(\cos\alpha, \cos\beta)$.

The contribution of a single point- or directional light source with intensity I_i to the intensity of the surface is given as $I_o = f_r(\vec{l} \to \vec{v}) \cos\alpha \cdot I_i$. The term $f_r(\mathbf{x}, \vec{l} \to \vec{v}) \cos\alpha$ can be split into two bivariate parts $F(\cos\theta) \cdot D(\cos\delta)$ and $G(\cos\alpha, \cos\beta)/(\pi \cdot \cos\beta)$, which are then stored in two independent 2-dimensional lookup tables.

Regular 2D texture mapping can be used to implement the lookup process. If all vectors are normalized, the texture coordinates are simple dot products between the surface normal, the viewing and light directions, and the micro facet normal. These vectors and their dot products can be computed in software and assigned as texture coordinates to each vertex of the object.

The interpolation of these texture coordinates across a polygon corresponds to a linear interpolation of the vectors without renormalization. Since the reflection model itself is highly nonlinear, this is much better than simple Gouraud shading, but not as good as evaluating the illumination in every pixel (Phong shading). The interpolation of normals without renormalization is commonly known as *fast Phong shading*.

This method for looking up the illumination in two separate 2-dimensional textures requires either a single rendering pass with two simultaneous textures, or two separate rendering passes with one texture each in order to render specular reflections on an object. If two passes are used, their results are multiplied using alpha blending. A third rendering pass with hardware lighting (or a third simultaneous texture) is applied for adding a diffuse term.

If the light and viewing directions are assumed to be constant, that is, if a directional light and an orthographic camera are assumed, the computation of the texture coordinates can even be done in hardware. To this end, light and viewing direction as well as the halfway vector between them are used as row vectors in the texture matrix for the two textures:

$$\begin{bmatrix} 0 & 0 & 0 & \cos\theta \\ h_x & h_y & h_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \cos\delta \\ 0 \\ 1 \end{bmatrix} \quad (2)$$

$$\begin{bmatrix} l_x & l_y & l_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha \\ \cos\beta \\ 0 \\ 1 \end{bmatrix} \quad (3)$$

Figure 2 shows a torus rendered with two different roughness settings using this technique.

We would like to note that the use of textures for representing the lighting model introduces an approximation error: while the term $F \cdot D$ is bounded by the interval $[0, 1]$, the second term $G/(\pi \cdot \cos\beta)$ exhibits a singularity for grazing viewing directions ($\cos\beta \to 0$). Since graphics hardware typically uses a fixed-point representation of textures, the texture values are clamped to the range $[0, 1]$. When these clamped values are used for the illumination process, areas around the grazing angles can be rendered too dark, especially if the surface is very shiny. This artifact can be reduced by dividing the values stored in the texture by a constant which is later multiplied back onto the final result. In practice, however, these artifacts are hardly noticeable.

The same methods can be applied to all kinds of variations of the Torrance-Sparrow model, using different distribution functions and geometry terms, or the approximations proposed in [24]. With varying numbers of terms and rendering passes, it is also possible to come up with similar factorizations for all kinds of other models. For example the Phong, Blinn-Phong

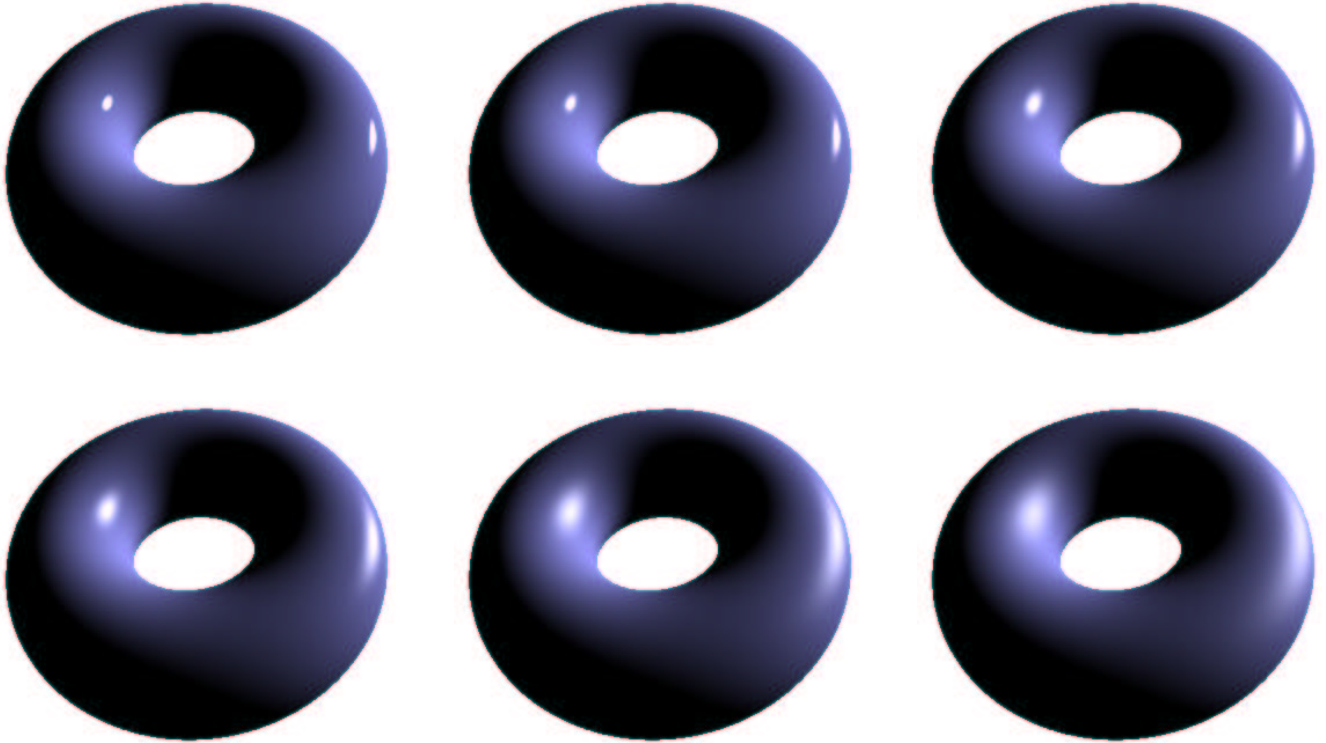

Figure 2: A torus rendered with the proposed hardware multi-pass method using the Torrance-Sparrow reflection model (Gaussian height distribution and geometry term by [27]) and different settings for the surface roughness. For these images, the torus was tessellated into 200×200 polygons.

and Cosine Lobe models can all be rendered in a single pass with a single texture, which can even already account for an ambient and a diffuse term in addition to the specular one.

1.1.1 Anisotropy

Although the treatment of anisotropic materials is somewhat harder, similar factorization techniques can be applied here. For anisotropic models, the micro facet distribution function and the geometrical attenuation factor also depend on the angle ϕ between the facet normal and a reference direction in the tangent plane. This reference direction is given in the form of a tangent vector $\vec{t}$.

For example, the elliptical Gaussian model [31] introduces an anisotropic facet distribution function specified as the product of two independent Gaussian functions, one in the direction of $\vec{t}$, and one in the direction of the binormal $\vec{n} \times \vec{t}$. This makes D a bivariate function in the angles δ and ϕ. Consequently, the texture coordinates can be computed in software in much the same way as described above for isotropic materials. This also holds for the other anisotropic models in computer graphics literature.

Since anisotropic models depend on both a normal and a tangent per vertex, the texture coordinates cannot be generated with the help of a texture matrix, even if light and viewing directions are assumed to be constant. This is due to the fact that the anisotropic term can usually not be factored into a term that only depends on the surface normal, and one that only depends on the tangent.

One exception to this rule is the model by Banks [2], which is mentioned here despite the fact that it is an *ad-hoc* model which is not based on physical considerations. Banks defines the reflection off an anisotropic surface as

$$I_o = \cos\alpha \cdot (k_d \langle \vec{n'} | \vec{l} \rangle + k_s \langle \vec{n'} | \vec{h} \rangle^{1/r}) \cdot I_i, \qquad (4)$$

where $\vec{n'}$ is the projection of the light vector $\vec{l}$ into the

plane perpendicular to the tangent vector $\vec{t}$. This vector is then used as a shading normal for a Blinn-Phong lighting model with diffuse and specular coefficients k_d and k_s, and surface roughness r. In [28], it has been pointed out that this Phong term is really only a function of the two angles between the tangent and the light direction, as well as the tangent and the viewing direction. This fact was used for the illumination of lines in [28].

Applied to anisotropic reflection models, this means that this Phong term can be looked up from a 2-dimensional texture, if the tangent $\vec{t}$ is specified as a texture coordinate, and the texture matrix is set up as in Equation 3. The additional term $\cos\alpha$ in Equation 4 is computed by hardware lighting with a directional light source and a purely diffuse material, so that the Banks model can be rendered with one texture and one pass per light source. Figure 3 shows images rendered with this reflection model.

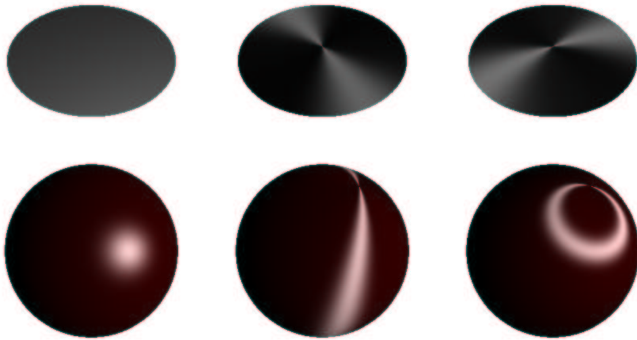

Figure 3: Disk and sphere illuminated with isotropic reflection (left), anisotropic reflection with circular features (center), and radial features (right).

1.1.2 Measured or Simulated Data

As mentioned above, the idea of factorizing BRDFs into low-dimensional parts that can be sampled and stored as textures not only applies to analytical reflection models, but also to BRDFs given in a tabular form. Different numerical methods have been presented for factorizing these tabular BRDFs [12, 18]. The discussion of these is beyond the scope of this course, however.

The advantage of the analytical factorization is that it is very efficient to adjust parameters of the reflection model, so this can be done interactively. The numerical methods take too long for that. On the other hand, the big advantage of the numerical methods is that arbitrary BRDFs resulting from measurements or physical simulations can be used. Figure 4, for example, shows a teapot with a BRDF that looks blue from one side and red from another. This BRDF has been generated using a simulation of micrgeometry [8].

Figure 4: A teapot with a simulated BRDF.

1.2 Global Illumination using Environment Maps

The presented techniques for applying alternative reflection models to local illumination computations can significantly increase the realism of synthetic images. However, true photorealism is only possible if global effects are also considered. Since texture mapping techniques for diffuse illumination are widely known and applied, we concentrate on non-diffuse global illumination, in particular mirror- and glossy reflection.

We describe here an approach based on environment maps, as presented by Heidrich and Seidel [9], because they offer a good compromise between rendering quality and storage requirements. With environment maps, 2-dimensional textures instead of the full 4-dimensional radiance field [19] can be used to store the illumination.

1.3 View-independent Environment Maps

The techniques described in the following assume that environment maps can be reused for different viewing positions in different frames, once they have been generated. It is therefore necessary to choose a representation for environment maps which is valid for arbitrary viewing positions. This includes both cube maps [6]

and parabolic maps [9], both of which are supported on all modern platforms.

1.4 Mirror and Diffuse Terms with Environment Maps

Once an environment map is given in a view-independent parameterization, it can be used to add a mirror reflection term to an object. Using multi-pass rendering and either alpha blending or an accumulation buffer [7], it is possible to add a diffuse global illumination term through the use of a precomputed texture. Two methods exist for the generation of such a texture. One way is, that a global illumination algorithm such as Radiosity is used to compute the diffuse global illumination in every surface point.

The second approach is purely image-based, and was proposed by Greene [6]. The environment map used for the mirror term contains information about the incoming radiance $L_i(\mathbf{x}, \vec{l})$, where $\mathbf{x}$ is the point for which the environment map is valid, and $\vec{l}$ the direction of the incoming light. This information can be used to prefilter the environment map to represent the diffuse reflection of an object for all possible surface normals. Like regular environment maps, this texture is only valid for one point in space, but can be used as an approximation for nearby points.

1.5 Fresnel Term

A regular environment map without prefiltering describes the incoming illumination in a point in space. If this information is directly used as the outgoing illumination, as with regular environment mapping, only metallic surfaces can be modeled. This is because for metallic surfaces (surfaces with a high index of refraction) the Fresnel term is almost one, independent of the angle between light direction and surface normal. Thus, for a perfectly smooth (i.e. mirroring) surface, incoming light is reflected in the mirror direction with a constant reflectance.

For non-metallic materials (materials with a small index of refraction), however, the reflectance strongly depends on the angle of the incoming light. Mirror reflections on these materials should be weighted by the Fresnel term for the angle between the normal and the viewing direction $\vec{v}$.

Similar to the techniques for local illumination presented in Section 1, the Fresnel term $F(\cos\theta)$ for the mirror direction $\vec{r}_v$ can be stored in a texture map. Since here only the Fresnel term is required, a 1-dimensional texture map suffices for this purpose. This Fresnel term is rendered to the framebuffer's alpha channel in a separate rendering pass. The mirror part is then multiplied with this term in a second pass, and a third pass is used to add the diffuse part. This yields an outgoing radiance of $L_o = F \cdot L_m + L_d$, where L_m is the contribution of the mirror term, while L_d is the contribution due to diffuse reflections.

In addition to simply adding the diffuse part to the Fresnel-weighted mirror reflection, we can also use the Fresnel term for blending between diffuse and specular: $L_o = F \cdot L_m + (1 - F)L_d$. This allows us to simulate diffuse surfaces with a transparent coating: the mirror term describes the reflection off the coating. Only light not reflected by the coating hits the underlying surface and is there reflected diffusely.

Figure 5 shows images generated using these two approaches. In the top row, the diffuse term is simply added to the Fresnel-weighted mirror term (the glossy reflection is zero). For a refractive index of 1.5 (left), which approximately corresponds to glass, the object is only specular for grazing viewing angles, while for a high index of refraction (200, right image), which is typical for metals, the whole object is highly specular.

The bottom row of Figure 5 shows two images generated with the second approach. For a low index of refraction, the specular term is again high only for grazing angles, but in contrast to the image above, the diffuse part fades out for these angles. For a high index of refraction, which, as pointed out above, corresponds to metal, the diffuse part is practically zero everywhere, so that the object is a perfect mirror for all directions.

1.6 Precomputed Glossy Reflection and Transmission

We would now like to extend the concept of environment maps to glossy reflections. The idea is similar to the diffuse prefiltering proposed by Greene [6] and the approach by Voorhies and Foran [30] to use environment maps to generate Phong highlights from directional light sources. These two ideas can be combined to precompute an environment map containing the glossy reflection of an object with a Phong material. With this concept, effects similar to the ones presented by Debevec [5] are possible in real time.

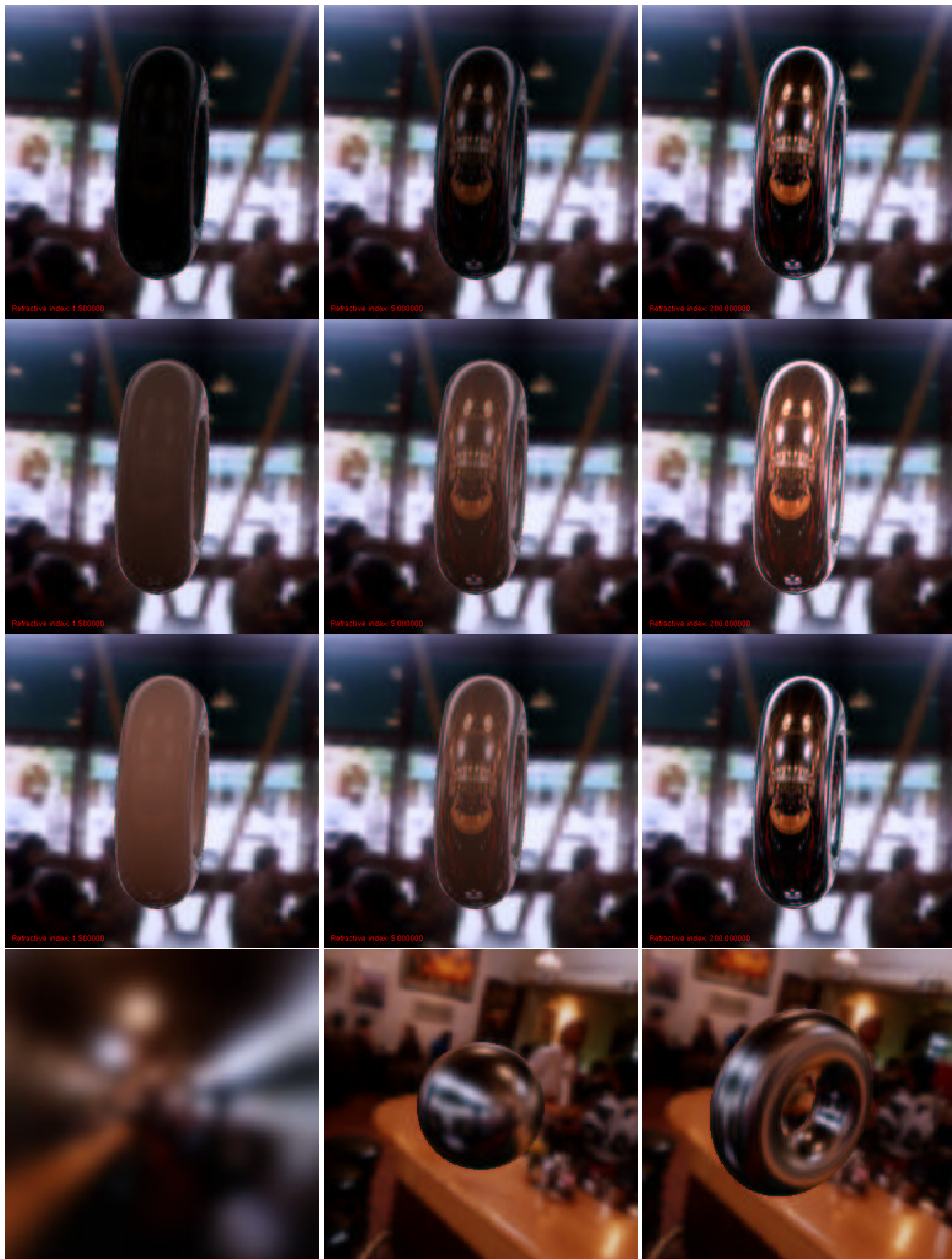

Figure 5: Top row: Fresnel weighted mirror term. Second row: Fresnel weighted mirror term plus diffuse illumination. Third row: Fresnel blending between mirror and diffuse term. The indices of refraction are (from left to right) 1.5, 5, and 200. Bottom row: a prefiltered version of the map with a roughness of 0.01, and application of this map to a reflective sphere and torus.

As shown in [15], the Phong BRDF is given by

$$f_r(\vec{l} \rightarrow \vec{v}) = k_s \cdot \frac{\langle \vec{r}_l | \vec{v} \rangle^{1/r}}{\cos\alpha} = k_s \cdot \frac{\langle \vec{r}_v | \vec{l} \rangle^{1/r}}{\cos\alpha}, \qquad (5)$$

where $\vec{r}_l$, and $\vec{r}_v$ are the reflected light- and viewing directions, respectively.

Thus, the specular global illumination using the Phong model is

$$L_o(\vec{r}_v) = k_s \cdot \int_{\Omega(\vec{n})} \langle \vec{r}_v | \vec{l} \rangle^{1/r} L_i(\vec{l}) \, d\omega(\vec{l}), \qquad (6)$$

which is only a function of the reflection vector $\vec{r}_v$ and the environment map containing the *incoming* radiance $L_i(\vec{l})$. Therefore, it is possible to take a map containing $L_i(\vec{l})$, and generate a filtered map containing the *outgoing* radiance for a glossy Phong material. Since this filtering is relatively expensive, it can on most platforms not be redone for every frame in an interactive application. On special graphics hardware that supports convolution operations, however, it can be performed on the fly, as described by Kautz et al. [13].

The bottom row of Figure 5 shows such a prefiltered map as well as applications of this map for reflection and transmision. If the original environment map is given in a high-dynamic range format, then this prefiltering technique allows for effects similar to the ones described by Debevec [5].

2 Shadow Mapping

After discussing models for local illumination in the previous chapter, we now turn to global effects. In this chapter we deal with algorithms for generating shadows in hardware-based renderings.

Shadows are probably the visually most important global effect. This fact has resulted in a lot of research on how to generate them in hardware-based systems. Thus, interactive shadows are in principle a solved problem. However, current graphics hardware rarely directly supports shadows, and, as a consequence, fewer applications than one might expect actually use the developed methods.

In contrast to the analytic approach shadow volumes, shadow maps [33] are a sampling-based method. First, the scene is rendered from the position of the light source, using a virtual image plane (see Figure 6). The depth image stored in the z-buffer is then used to test whether a point is in shadow or not.

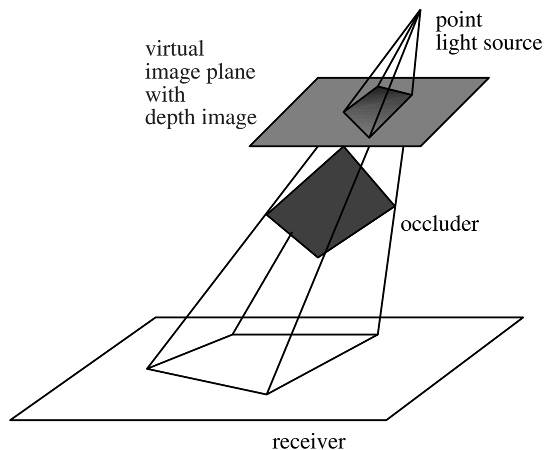

Figure 6: Shadow maps use the z-buffer of an image of the scene rendered from the light source.

To this end, each fragment as seen from the camera needs to be projected onto the depth image of the light source. If the distance of the fragment to the light source is equal to the depth stored for the respective pixel, then the fragment is lit. If the fragment is further away, is is in shadow.

A hardware multi-pass implementation of this principle has been proposed in [25]. The first step is the acquisition of the shadow map by rendering the scene from the light source position. For walkthroughs, this is a preprocessing step, for dynamic scenes it needs to be performed each frame. Then, for each frame, the scene is rendered without the illumination contribution from the light source. In a second rendering pass, the shadow map is specified as a projective texture, and a specific hardware extension is used to map each pixel into the local coordinate space of the light source and perform the depth comparison. Pixels passing this depth test are marked in the stencil buffer. Finally, the illumination contribution of the light source is added to the lit regions by a third rendering pass.

The advantage of the shadow map algorithm is that it is a general method for computing all shadows in the scene, and that it is very fast, since the representation of the shadows is independent of the scene complexity. On the down side, there are artifacts due to the discrete sampling and the quantization of the depth. One benefit of the shadow map algorithm is that the rendering quality scales with the available hardware. The method could be implemented on fairly low end systems, but for high end systems a higher resolution or deeper z-buffer could be chosen, so that the quality in-

creases with the available texture memory. Unfortunately, the necessary hardware extensions to perform the depth comparison on a per-fragment basis are currently only availablehave until recently only been available on two high-end systems, the RealityEngine [1] and the InfiniteReality [20].

2.1 Shadow Maps Using the Alpha Test

Instead of relying on a dedicated shadow map extension, it is also possible to use projective textures and the alpha test. Basically, this method is similar to the method described in [25], but it efficiently takes advantage of automatic texture coordinate generation and the alpha test to generate shadow masks on a per-pixel basis. This method takes one rendering pass more than required with the appropriate hardware extension.

In contrast to traditional shadow maps, which use the contents of a z-buffer for the depth comparison, we use a depth map with a *linear* mapping of the z values in light source coordinates. This allows us to compute the depth values via automatic texture coordinate generation instead of a per-pixel division. Moreover, this choice improves the quality of the depth comparison, because the depth range is sampled uniformly, while a z-buffer represents close points with higher accuracy than far points.

As before, the entire scene is rendered from the light source position in a first pass. Automatic texture coordinate generation is used to set the texture coordinate of each vertex to the depth as seen from the light source, and a 1-dimensional texture is used to define a linear mapping of this depth to alpha values. Since the alpha values are restricted to the range $[0 \dots 1]$, near and far planes have to be selected, whose depths are then mapped to alpha values 0 and 1, respectively. The result of this is an image in which the red, green, and blue channels have arbitrary values, but the alpha channel stores the depth information of the scene as seen from the light source. This image can later be used as a texture.

For all object points visible from the camera, the shadow map algorithm now requires a comparison of the point's depth with respect to the light source with the corresponding depth value from the shadow map. The first of these two values can be obtained by applying the same 1-dimensional texture that was used for generating the shadow map. The second value is obtained simply by using the shadow map as a projective texture. In order to compare the two values, we can subtract them from each other, and compare the result to zero.

With multi-texturing, this comparison can be implemented in a single rendering pass. Both the 1-dimensional texture and the shadow map are specified as simultaneous textures, and the texture blending function is used to implement the difference. The resulting α value is 0 at each fragment that is lit by the light source, and > 0 for fragments that are shadowed. Then, an alpha test is employed to compare the results to zero. Pixels passing the alpha test are marked in the stencil buffer, so that the lit regions can then be rendered in a final rendering pass.

Without support for multi-texturing, the same algorithm is much more expensive. First, two separate passes are required for applying the texture maps, and alpha blending is used for the difference. Now, the framebuffer contains an α value of 0 at each pixel that is lit by the light source, and > 0 for shadowed pixels. In the next step it is then necessary to set α to 1 for all the shadowed pixels. This will allow us to render the lit geometry, and simply multiply each fragment by $1-\alpha$ of the corresponding pixel in the framebuffer (the value of $1 - \alpha$ would be 0 for shadowed and 1 for lit regions). In order to do this, we have to copy the framebuffer onto itself, thereby scaling α by 2^n, where n is the number of bits in the α channel. This ensures that $1/2^n$, the smallest value > 0, will be mapped to 1. Due to the automatic clamping to the interval $[0 \dots 1]$, all larger values will also be mapped to 1, while zero values remain zero. In addition to requiring an expensive framebuffer copy, this algorithm also needs an alpha channel in the framebuffer ("destination alpha"), which might not be available on some systems.

Figure 7 shows an engine block where the shadow regions have been determined using this approach. Since the scene is rendered at least three times for every frame (four times if the light source or any of the objects move), the rendering times for this method strongly depend on the complexity of the visible geometry in every frame, but not at all on the complexity of the geometry casting the shadows. Scenes of moderate complexity can be rendered at high frame rates even on low end systems. The images in Figure 7 are actually the results of texture-based volume rendering using 3D texturing hardware (see [32] for the details of the illumination process).

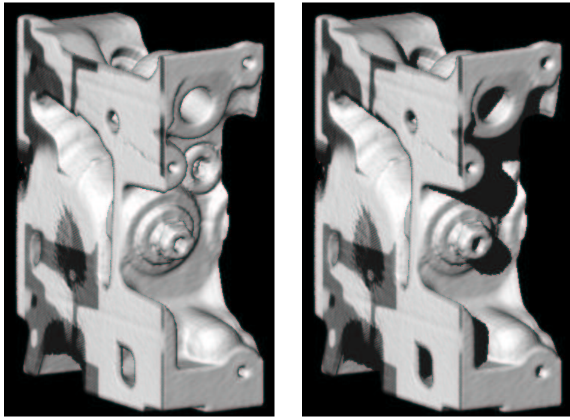

Figure 7: An engine block generated from a volume data set with and without shadows. The shadows have been computed with our algorithm for alpha-coded shadow maps. The Phong reflection model is used for the unshadowed parts.

3 Bump Mapping Algorithms

Bump maps have become a popular approach for adding visual complexity to a scene, without increasing the geometric complexity. They have been used in software rendering systems for quite a while [4], but hardware implementations have only occurred relatively recently, and several different methods are possible, depending on the level of hardware support (e.g. [23, 22, 9, 14]).

The original approach to bump mapping [4] defines surface detail as a height value at every point on a smooth base surface. From this texture-mapped height value, one can compute a per-pixel normal by taking the partial derivatives of the height values. Since this is a fairly expensive operation, most recent hardware implementations [22, 9, 14] precompute the normal for every surface point in an offline process, and store it directly in a texture map.

The bump mapping scheme that has become most popular for interactive applications recently is described in detail in a technical report by Kilgard [14]. First, the light and the viewing vector at every vertex of the geometry is computed and transformed into the local coordinate frame at that vertex ("tangent space", see [22]). In the original version, this is a software step, which can now, however also be done directly in hardware [16]. Then, these local vectors are interpolated across the surface using Gouraud shading and the per-pixel bump map normals are looked up from a texture map. A simple reflection model containing a diffuse and a Phong component can then be implemented as a number of dot products followed by successive squaring (for the Phong exponent). These operations map easily to the register combiner facility present in modern hardware [21].

3.1 Shadows for Bump Maps

The basic approach to bump mapping as outlined above can be extended to approximate the shadows that the bumps cast onto each other. Note that approaches like shadow maps do not work for bump maps because during the rendering phase the geometry is not available; only per-pixel normals are. Shadowing algorithms for bump maps therefore encode the visibility of every surface point for every possible light direction. This is simplified by the fact that bump maps are derived from height fields (i.e. terrains), which allows us to use the notion of a *horizon*. In a terrain, a distant light source located in a certain direction is visible from a given surface point if and only if it is located above the horizon for that surface point. Thus, it is sufficient to encode the horizon for all height field points and directions. This approach is called *horizon mapping*, first presented by Max [17].

The question is, how this horizon information can be represented such that it consumes little memory, and such that the test of whether a given light direction is above or below the horizon for any point in the bump map can be done efficiently in hardware. We describe here a method proposed by Heidrich et al. [8].

We start with a bump map given as a height field, as in the original formulation by Blinn [4]. We then select a number of random directions $D = \{d_i\}$, and shoot rays from all height field points $\mathbf{p}$ into each of the directions d_i. For the shadowing algorithm we will only record a boolean value for each of these rays, namely whether the ray hits another point in the height field, or not. In Section 3.2 we will describe how to use a similar preprocessing step for computing indirect illumination in bump maps.

Now let us consider all the rays shot from a single surface point $\mathbf{p}$. We project all the unit vectors for the sampling directions $\vec{d_i} \in D$ into the tangent plane, i.e. we drop the z coordinate of $\vec{d_i}$ in the local coordinate frame. Then we fit an ellipse containing as many of those 2D points that correspond to unshadowed directions as possible, without containing too many shad-

owed directions. This ellipse is uniquely determined by its (2D) center point **c**, a direction $(a_x, a_y)^T$ describing the direction of the major axis (the minor axis is then simply $(-a_y, a_x)^T$), and two radii r_1 and r_2, one for the extent along each axis.

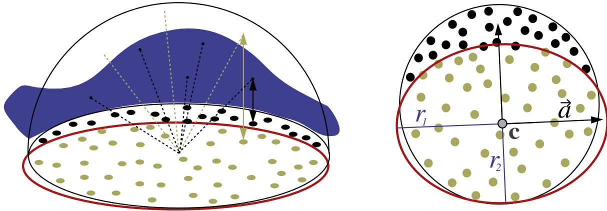

Figure 8: *For the shadow test we precompute 2D ellipses at each point of the height field, by fitting them to the projections of the scattering directions into the tangent plane.*

For the fitting process, we begin with the ellipse represented by the eigenvectors of the covariance matrix of all points corresponding to unshadowed directions. We then optimize the radii with a local optimization method. As an optimization criterion we try to maximize the number of light directions inside the ellipse while at the same time minimizing the number of shadowed directions inside it.

Once we have computed this ellipse for each grid point in the height field, the shadow test is simple. The light direction $\vec{l}$ is also projected into the tangent plane, and it is checked whether the resulting 2D point is inside the ellipse (corresponding to a lit point) or not (corresponding to a shadowed point).

Both the projection and the in-ellipse test can mathematically be expressed very easily. First, the 2D coordinates l_x and l_y have to be transformed into the coordinate system defined by the axes of the ellipse:

$$l'_x := \langle \begin{pmatrix} a_x \\ a_y \end{pmatrix} \mid \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle, \tag{7}$$

$$l'_y := \langle \begin{pmatrix} -a_y \\ a_x \end{pmatrix} \mid \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle \tag{8}$$

Afterwards, the test

$$1 - \frac{(l'_x)^2}{r_1^2} - \frac{(l'_y)^2}{r_2^2} \geq 0 \tag{9}$$

has to be performed.

To map these computations to graphics hardware, we represent the six degrees of freedom for the ellipses as 2 RGB textures. Then the required operations to implement Equations 7 through 9 are simple dot products as well as additions and multiplications. This is possible using the OpenGL imaging subset [26], available on most contemporary workstations, but also using some vendor specific extensions, such as the *register combiner* extension from NVIDIA [21]. Depending on the exact graphics hardware available, the implementation details will have to vary slightly. These details for different platforms are described in a technical report [11].

Figure 9 shows some results of this shadowing algorithm.

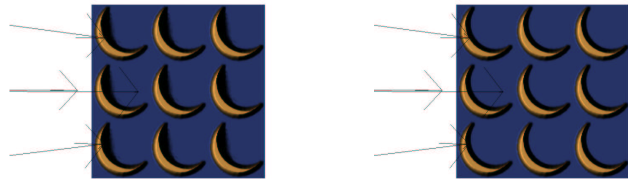

Figure 9: A simple bump map with and without shadows

3.2 Indirect Illumination in Bump Maps

Finally, we would like to discuss a method for computing the indirect light in bump maps [8], i.e. the light that bounces around multiple times in the bumps before hitting the camera.

As in the case of bump map shadows, we start by choosing a set of random directions $d_i \in D$, and shooting rays from al points **p** on the height field into all directions d_i. This time, however, we do not only store a boolean value for every ray, but rather the 2D coordinates of the intersection of that ray with the height field (if any). That is, for every direction d_i, we store a 2D map S_i that, for every point **p**, holds the 2D coordinates of the point **q** visible from **p** in direction d_i.

Using this precomputed visiblity information, we can then integrate over the light arriving from all directions. For every point **p** in the height field, we sum up the indirect illumination arriving from any of the directions d_i, as depicted in Figure 10.

If we assume that both the light and the viewing direction vary slowly across the height field (this corresponds to the assumption that the bumps are relatively small compared to the distance from both the viewer and the light source), then the only strongly varying pa-

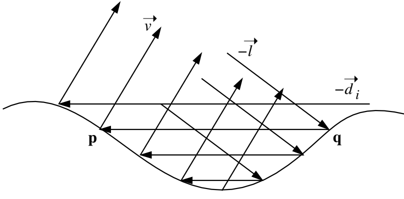

Figure 10: *With the precomputed visibility, the different paths for the illumination in all surface points are composed of pieces with identical directions.*

rameters are the surface normals. More specifically, for the radiance leaving a grid point $\mathbf{p}$ in direction $\vec{v}$, the important varying parameters are the normal $\vec{n}_p$, the point $\mathbf{q} := S_i[\mathbf{p}]$ visible from $\mathbf{p}$ in direction $\vec{d}_i$, and the normal $\vec{n}_q$ in that point.

In particular, the radiance in direction $\vec{v}$ caused by light arriving from direction $\vec{l}$ and scattered once in direction $-\vec{d}_i$ is given by the following formula.

$$L_o(\mathbf{p}, \vec{v}) = f_r(\vec{n}_p, \vec{d}_i, \vec{v})\langle\vec{n}_p|\vec{d}_i\rangle \cdot \left(f_r(\vec{n}_q, \vec{l}, -\vec{d}_i)\langle\vec{n}_q|\vec{l}\rangle \cdot L_i(\mathbf{q}, \vec{l})\right). \quad (10)$$

Usually, the BRDF is written as a 4D function of the incoming and the outgoing direction, both given relative to a local coordinate frame where the local surface normal coincides with the z-axis. In a height field setting, however, the viewing and light directions are given in some global coordinate system that is not aligned with the local coordinate frame, so that it is first necessary to perform a transformation between the two frames. To emphasize this fact, we have denoted the BRDF as a function of the incoming and outgoing direction as well as the surface normal. If we plan to use an anisotropic BRDF on the micro geometry level, we would also have to include a reference tangent vector.

Note that the term in parenthesis is simply the direct illumination of a height field with viewing direction $-\vec{d}_i$, with light arriving from $\vec{l}$. If we precompute this term for all grid points in the height field, we obtain a texture L_d containing the direct illumination for each surface point. This texture can be generated using a bump mapping step where an orthographic camera points down onto the height field, but $-\vec{d}_i$ is used as the viewing direction for shading purposes.

Once we have L_d, the second reflection is just another bump mapping step with $\vec{v}$ as the viewing direction and $\vec{d}_i$ as the light direction. This time, the incoming radiance is not determined by the intensity of the light source, but rather by the content of the L_d texture. For each surface point $\mathbf{p}$ we look up the corresponding visible point $\mathbf{q} = S_i[\mathbf{p}]$. The outgoing radiance at $\mathbf{q}$, which is stored in the texture as $L_d[\mathbf{q}]$, is at the same time the incoming radiance at $\mathbf{p}$.

Thus, we have reduced computing the once-scattered light in each point of the height field to two successive bump mapping operations, where the second one requires an additional indirection to look up the illumination. We can easily extend this technique to longer paths, and also add in the direct term at each scattering point. This is illustrated in the Figure 11.

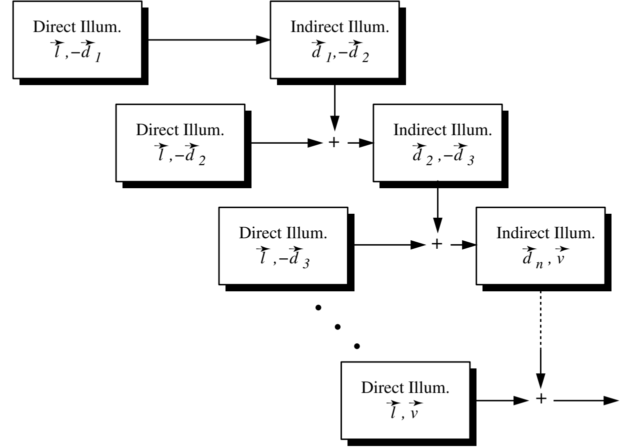

Figure 11: *Extending the dependent test scattering algorithm to multiple scattering. Each box indicates a texture that is generated with regular bump mapping.*

For the total illumination in a height field, we sum up the contributions for several such paths (some 40-100 in most of our scenes). This way, we compute the illumination in the complete height field at once, using two SIMD-style operations on the whole height field texture: bump mapping for direct illumination, using two given directions for incoming and outgoing light, as well as a lookup of the indirect illumination in a texture map using the precomputed visibility data in form of the textures S_i.

3.2.1 Use of Graphics Hardware

In recent graphics hardware, both on the workstation and on the consumer level, several new features have been introduced that we can make use of. In particular, we assume a standard OpenGL-like graphics pipeline [26] with some extensions as described in the following.

Firstly, we assume the hardware has some way of rendering bump maps. This can either be supported

through specific extensions (e.g. [21]), or through the OpenGL imaging subset [26], as described by Heidrich and Seidel [9]. Any kind of bump mapping scheme will be sufficient for our purposes, but the kind of reflection model available in this bump mapping step will determine what reflection model we can use to illuminate our hight field.

Secondly, we will need a way of interpreting the components stored in one texture or image as texture coordinates pointing into another texture. One way of supporting this is the so-called *pixel texture* extension [10, 9], which performs this operation during transfer of images into the frame buffer, and is currently only available on some high-end SGI machines. Alternatively, we can use *dependent texture lookups*, a variant of multi-texturing, that has recently become available on some newer PC graphics boards. With dependent texturing, we can map two or more textures simultaneously onto an object, where the texture coordinates of the second texture are obtained from the components of the first texture. This is exactly the feature we are looking for. In case we have hardware that supports neither of the two, it is quite simple, although not very fast, to implement the pixel texture extension in software: the framebuffer is read out to main memory, and each pixel is replaced by a value looked up from a texture, using the previous contents of the pixel as texture coordinates.

Using these two features, dependent texturing and bump mapping, the implementation of the dependent test method as described above is simple. As depicted in Figure 10, the scattering of light via two points $\mathbf{p}$ and $\mathbf{q}$ in the height field first requires us to compute the direct illumination in $\mathbf{q}$. If we do this for all grid points we obtain a texture L_d containing the reflected light caused by the direct illumination in each point. This texture L_d is generated using the bump mapping mechanism the hardware provides. Typically, the hardware will support only diffuse and Phong reflections, but if it supports more general models, then these can also be used for our scattering implementation.

The second reflection in $\mathbf{p}$ is also a bump mapping step (although with different viewing- and light directions), but this time the direct illumination from the light source has to be replaced by a per-pixel radiance value corresponding to the reflected radiance of the point $\mathbf{q}$ visible from $\mathbf{p}$ in the scattering direction. We achieve this by bump mapping the surface with a light intensity of 1, and by afterwards applying a pixel-wise multiplication of the value looked up from L_d with the help of dependent texturing. Figure 12 shows how to conceptually set up a multi-texturing system with dependent textures to achieve this result.

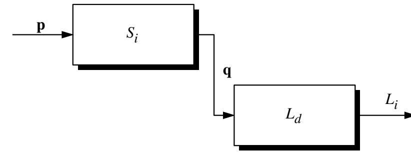

Figure 12: *For computing the indirect light with the help of graphics hardware, we conceptually require a multi-texturing system with dependent texture lookups. This figure illustrates how this system has to be set up. Boxes indicate one of the two textures, while incoming arrows signal texture coordinates and outgoing ones mean the resulting color values.*

The first texture is the S_i that corresponds to the scattering direction d_i. For each point $\mathbf{p}$ it yields $\mathbf{q}$, the point visible from $\mathbf{p}$ in direction d_i. The second texture L_d contains the reflected direct light in each point, which acts as an incoming radiance at $\mathbf{p}$. Figure 13 shows some results of the method.

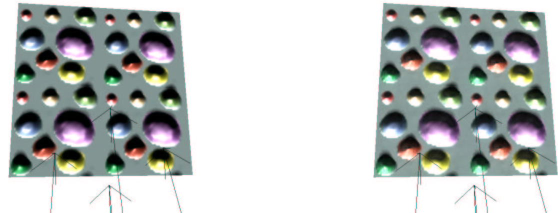

Figure 13: A bump map with and without indirect illumination

By using this hardware approach, we treat the graphics board as a SIMD-like machine which performs the desired operations, and computes one light path for each of the grid points at once. This use of hardware dramatically increases the performance over the software version to an almost interactive rate.

4 Conclusion

In this part, we have reviewed some of the more complex shading algorithms that utilize graphics hardware. While the individual methods are certainly quite different, there are some features that occur in all examples:

- The most expensive operations (i.e. visibility computations, filtering of environment maps etc.) are not performed on the fly, but are done in a precomputing step.
- The results of the precomputation are represented in a sampled (tabular) form that allows us to use texture mapping to apply the information in the actual shaders.
- The shaders themselves are often relatively simple due to the amount of precomputation. They mostly have the job of combining the precomputed textures in various flexible ways.
- The textures need to be parameterized in such a way that the texture coordinates are easy and efficient to generate, ideally directly in hardware.

References

[1] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, August 1993.

[2] David C. Banks. Illumination in diverse codimensions. In *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 327–334, July 1994.

[3] Petr Beckmann and Andre Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. McMillan, 1963.

[4] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, August 1978.

[5] Paul E. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 189–198, July 1998.

[6] Ned Greene. Applications of world projections. In *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[7] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 309–318, August 1990.

[8] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pages 455–464, July 2000.

[9] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August 1999.

[10] Silicon Graphics Inc. *Pixel Texture Extension*, December 1996. Specification document, available from http://www.opengl.org.

[11] Jan Kautz, Wolfgang Heidrich, and Katja Daubert. Bump map shadows for OpenGL rendering. Technical Report MPI-I-2000-4-001, Max-Planck-Institut für Informatik, 2000.

[12] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In *Rendering Techniques '99 (Proc. of Eurographics Workshop on Rendering)*, pages 247 – 260, June 1999.

[13] Jan Kautz, Pere-Pau Vázquez, Wolfgang Heidrich, and Hans-Peter Seidel. Unified approach to prefiltered environment maps. In *Rendering Techniques '00*.

[14] Mark Kilgard. A practical and robust bump mapping technique. Technical report, NVIDIA, 2000. available from http://www.nvidia.com.

[15] Robert R. Lewis. Making shaders more physically plausible. In *Fourth Eurographics Workshop on Rendering*, pages 47–62, June 1993.

[16] Erik Lindholm, Mark Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, August 2001.

[17] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[18] Anis Ahmad Michael D. McCool, Jason Ang. Homomorphic factorization of BRDFs for high-performance rendering. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, 2001.

[19] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 281–292, March 1998.

[20] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 293–302, August 1997.

[21] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, October 1999. Available from http://www.nvidia.com.

[22] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 303–306, August 1997.

[23] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, May 1996.

[24] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, pages 73–83, June 1993.

[25] Marc Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[26] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[27] Bruce G. Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967.

[28] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast display of illuminated field lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

[29] Kenneth E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, September 1967.

[30] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

[31] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 265–273, July 1992.

[32] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 169–178, July 1998.

[33] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.